Elmar Schömer
Ann-Christin Wörl

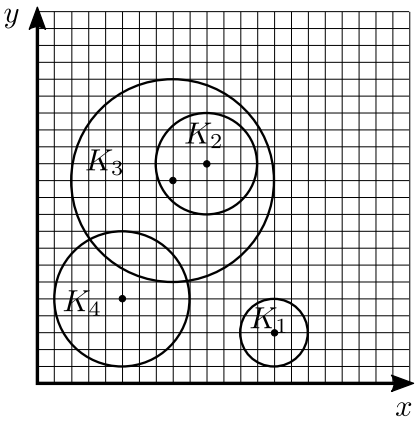
11. Übungsblatt

Abgabe: Dienstag, der 23.01.2024, 14:00 Uhr

Aufgabe 1: Kreise (5+5+5+5 Punkte)

Wir betrachten n Kreisscheiben. Eine Kreisscheibe K_i sei definiert durch ihren Mittelpunkt (x_i, y_i) und ihren Radius r_i. Alle Werte x_i, y_i und r_i seien ganze Zahlen.

$$K_i = \{(x, y) | (x - x_i)^2 + (y - y_i)^2 \leq r^2\}$$




```
class Kreis:
    ...

K1 = Kreis(14,3,2)
K2 = Kreis(10,13,3)
K3 = Kreis(8,12,6)
K4 = Kreis(5,5,4)

print(K1)
print(K1 in K3)
print(K2 in K3)
print(K2.schneidet(K3))
print(K4.schneidet(K3))


Kreis mit Mittelpunkt (14,3) und Radius 2
False
True
True
True
```

1. Begründen Sie die Korrektheit folgender Aussage: "Zwei Kreisscheiben K_i und K_j schneiden sich genau dann, wenn der Abstand ihrer Mittelpunkte kleiner oder gleich der Summe ihrer Radien ist." D.h.
$$K_i \cap K_j \neq \emptyset \iff (x_i - x_j)^2 + (y_i - y_j)^2 \leq (r_i + r_j)^2$$
Ergänzen Sie folgende Aussage: "Eine Kreisscheibe K_i liegt genau dann vollständig in der Kreisscheibe K_j, wenn ..."

2. Der oben stehende Python-Code erzeugt Kreisscheiben-Objekte und führt einige Operationen damit aus. Die Ausgabe der vier print-Anweisungen ist ebenfalls zu sehen. Ergänzen Sie den Code für die Klasse Kreis, sodass die gewünschte Funktionalität bereitgestellt wird. Den `in` Operator kann man überschreiben, indem man die Funktion `__contains__` neu definiert.

3. Gegeben sei ein Feld `K` von n Kreis-Objekten. Wir wollen diese Kreisscheiben nach ihrem Radius aufsteigend sortieren. Was ist dazu zu tun? Implementieren Sie den notwendigen Code in einer Funktion `sortieren(K)`.

4. Schreiben Sie eine Funktion `enthaelt(K)`, der man ein Feld `K` von Kreisscheiben übergibt und die eine zweidimensionale boolesche Matrix `E` erzeugt, aus der hervorgeht, ob `K[i]` in `K[j]` enthalten ist (`E[i][j] = True`) oder nicht (`E[i][j] = False`).

Aufgabe 2: Selbstorganisation (10+10+10 Punkte)

Wir wollen eine einfach verkettete Liste als das einfachste Beispiel einer sich selbst organisierenden Suchdatenstruktur betrachten (siehe https://en.wikipedia.org/wiki/Self-organizing_list). Die Suche nach einem Element in einer verketteten Liste dauert um so länger, je weiter hinten das Suchelement sich in der Liste befindet. Deshalb ist es sinnvoll, häufig nachgefragte Elemente weiter vorne in der Liste zu platzieren. In manchen Anwendungen ist die statistische Verteilung der Suchanfragen aber oft nicht bekannt und kann sich über die Zeit hin verändern. Wir wollen der Einfachheit halber eine invariante Verteilung von Suchanfragen nach den Elementen 0 bis 99 annehmen und zwei verschiedene Techniken der Selbstorganisation einer Liste implementieren und vergleichen.

1. Die Move-to-Front Regel: Das nachgefragte Element wird in der Liste lokalisiert und von seinem alten Ort an den Kopf der Liste bewegt.

2. Die Transpositionsregel: Das nachgefragte Element wird in der Liste lokalisiert und mit seinem direkten Vorgängerelement vertauscht, wenn es sich nicht schon am Kopf der Liste befindet.

Das untenstehende Programm baut eine initiale, einfach verkettete Liste von Elementen mit den Einträgen von 0 bis 99 auf und erzeugt beispielhaft eine kurze Folge von Suchanfragen nach einer festen Verteilung.

1. Implementieren Sie in der Klasse `Liste` die Move-to-Front-Regel.

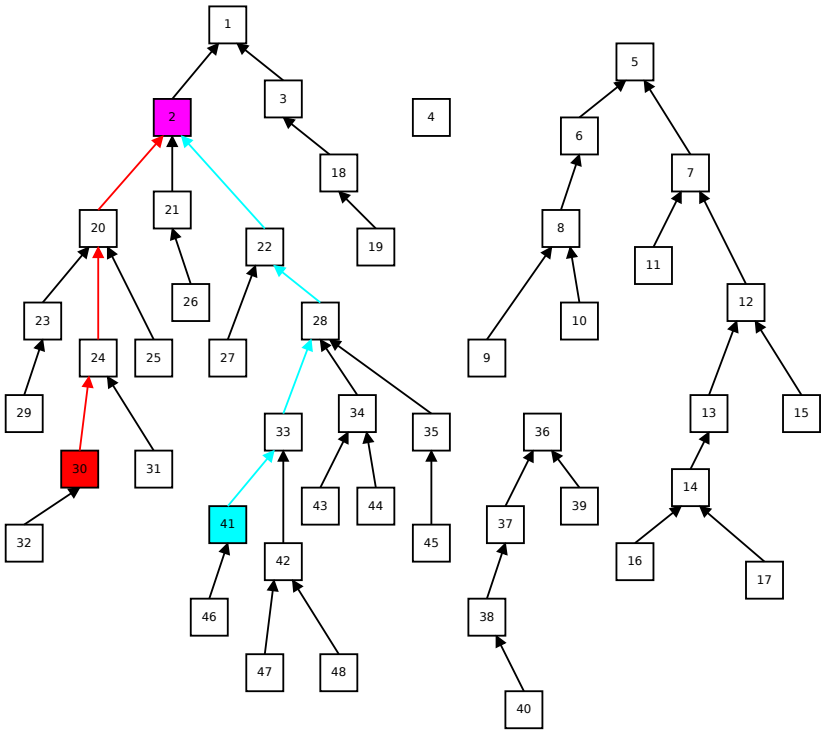
2. Implementieren Sie in der Klasse `Liste` die Transpositionsregel.

3. Untersuchen Sie experimentell, welche der beiden Regeln sich schneller an die gegebene Verteilung der Suchanfragen anpasst. Wie ordnen sich die Elemente im Laufe der Zeit an, wenn man deutlich mehr als 20 Suchanfragen wie im Beispielcode durchführt?

```
1  import random
2
3  class Listenknoten:
4      def __init__(self,inhalt):
5          self.inhalt = inhalt
6          self.next = None
7
8  class Liste:
9      def __init__(self):
10         self.kopf = None
11
12     def einfuege(self,inhalt):
13         knoten = Listenknoten(inhalt)
14         knoten.next = self.kopf
15         self.kopf = knoten
16
17     def suche(self,element):
18         knoten = self.kopf
19         while knoten != None:
20             if knoten.inhalt == element:
21                 return(True)
22             knoten = knoten.next
23         return(False)
24
25     def __str__(self):
26         s = ''
27         knoten = self.kopf
28         while knoten != None:
29             s += str(knoten.inhalt)+' '
30             knoten = knoten.next
31         return(s)
32
33 L = Liste()
34 for i in range(100):
35     L.einfuege(i)
36
37 print(L)
38
39 for i in range(20):
40     x = int(random.triangular(0,100,0))
41     print('suche Element '+str(x))
```

Aufgabe 3: Ahnentafel (5+5+5+5+5+10* Punkte)

Es gibt Erbsubstanz (siehe `https://de.wikipedia.org/wiki/Mitochondriale_DNA`), die in der Regel nur von der Mutter auf die Nachkommen vererbt wird. Aufgrund von Mutationen dieser Erbsubstanz kann man zum Beispiel den Verwandschaftsgrad von Volksstämmen untersuchen.



Wir betrachten eine Klasse `Frau`, die es erlaubt, die Abstammung einer Frau von Generation zu Generation zurückzuverfolgen. Dazu soll jede Frau eine Referenz auf ihre Mutter erhalten, wenn sie von ihr die besagte Erbsubstanz in exakter Kopie erhalten hat. Sollte eine Mutation aufgetreten sein, so wird die Referenz auf die Mutter auf `None` gesetzt.

```
1  class Frau:
2      def __init__(self,name):
3          self.name = name
4          self.mutter = None
5          # self.kinder = []
6          # self.markiert = False
```

1. In der Datei `data/Ahnen.txt` finden Sie die Ahnentafel zur obigen Grafik. In der ersten Zeile steht die Anzahl der betrachteten Frauen. Dann folgen die Kind-Mutter Beziehungen. Eine Zeile mit dem Eintrag u v bedeutet, dass v die Mutter von u ist. Wir wollen eine Klasse `Ahnentafel` zur Verwaltung aller Verwandschaftverhältnisse entwerfen. Erklären Sie die Funktionsweise des folgenden Konstruktors!

```
class Ahnentafel:
    def __init__(self,ahnentafel):
        datei = open(ahnentafel,'r')

        zeile = datei.readline()
        n = int(zeile)

        self.F = [Frau(i+1) for i in range(n)]

        for zeile in datei:
            s = zeile.split()
            kind = int(s[0])
            mutter = int(s[1])
            self.F[kind-1].mutter = self.F[mutter-1]
        datei.close()
```

2. Ergänzen Sie die beiden Klassen `Frau` und `Ahnentafel`, so dass zu jeder Frau auch die (weiblichen) Kinder abgespeichert werden.

3. Implementieren Sie eine Memberfunktion `anzahlMutanten`, die herhausfindet, wie viele Frauen in obigem Sinne keine Mutter haben, und die Anzahl zurückgibt.

4. Schreiben Sie eine (Member-)Funktion `zahlDerNachkommen`, die zu einer Frau die Zahl ihrer Nachkommen ermitteln kann.

5. Implementieren Sie außerdem eine Funktion `laengsteAhnenReihe()`, die die Länge der längsten Ahnenreihe ermittelt und zurückgibt.

6. **Zusatzaufgabe:** Implementieren Sie eine Funktion `juengsteGemeinsameAhnin(frau1,frau2)`, die für zwei Frauen die jüngste gemeinsame Vorfahrin ermittelt und zurückgibt. Falls diese nicht existiert, soll die Funktion `None` zurückgeben.

   Hinweis: Bei dieser Suche ist es hilfreich, eine Frau markieren zu können. Sie können dies bei der Definition der Klasse `Frau` berücksichtigen. Nach Aufruf der Funktion sollen jedoch alle Markierungen wieder entfernt worden sein, sodass die Funktion auch ein zweites Mal aufgerufen werden kann und korrekt funktioniert.

Aufgabe 4: Binäre Suchbäume (5+5+5+5+5+10* Punkte)

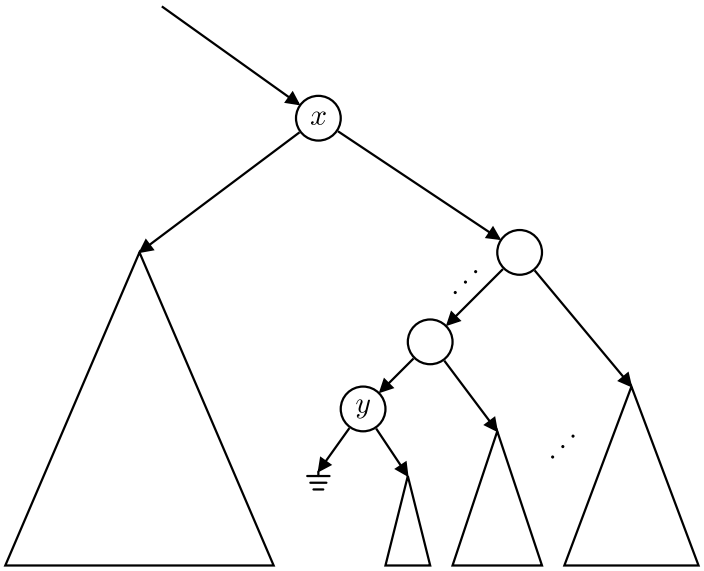
Wir wollen die Klassen `Baumknoten` und `Suchbaum` zur Realisierung eines binären Suchbaumes modifizieren beziehungsweise erweitern.

1. Ändern Sie die Memberfunktion `suche` so ab, dass sie keine Rekursion verwendet.

2. Überladen Sie die Zugriffsoperatoren `__getitem__` und `__setitem__`, sodass folgende Benutzung eines Suchbaumes mit dem Namen `S` möglich ist: `inhalt = S[schluessel]` und `S[schluessel] = neuer_inhalt`.

3. Ergänzen Sie eine Memberfunktion `first`, die den Knoten mit dem kleinsten Schlüssel im Suchbaum ermittelt.

4. Das folgende Diagramm zeigt den allgemeinen Fall, wie man ausgehend von einem Knoten mit Schlüssel x den Knoten mit dem nächst größeren Schlüssel y ermitteln kann. Beschreiben Sie in Worten, welchen Verweisen man vom Knoten x aus folgen muss, um den Knoten y zu erreichen. Welche Spezialfälle sind zu beachten? Wie sollte man vorgehen, wenn der Knoten mit dem Schlüssel x kein rechtes Kind besitzt?



5. Wie kann man vorgehen, wenn man den Knoten mit Schlüssel x löschen möchte? Eine Erklärung in Worten genügt.

   Hinweis: Damit der Baum durch die Löschung von x nicht zerfällt, kann man den Schlüssel x des betreffenden Knotens durch den Schlüssel y ersetzen und statt dessen den Knoten mit dem Schlüssel y löschen. Begründen Sie, warum dies korrekt ist und welche Verweise man dazu abändern muss.

6. **Zusatzaufgabe:** Implementieren Sie eine Memberfunktion `loesche`.